Webアプリケーション入門 🚀

バックエンドとデータベース

Webサイト・アプリの「3つの顔」

皆さんが見たり触ったりする部分は、実は全体の一部です。

1. **フロントエンド**: 見た目と操作する部分
2. **バックエンド**: 裏側で処理する部分
3. **データベース**: 情報をしまう倉庫

1. フロントエンド：お店の「見た目」と「入り口」

- 皆さんが**直接見る画面**（デザイン、ボタン、文字など）
- スマホやPCのブラウザで動く
- **HTML, CSS, JavaScript**などで作る
- 例：お店の窓、ドア、商品棚

2. バックエンド：お店の「キッチン」と「店員さん」

- 皆さんの**操作（リクエスト）を受け取る**裏側の部分
- サーバー（高性能なコンピューター）で動く
- **データ処理、計算、判断**をする
- **データベースとのやり取り**を担当
- **Ruby, Python, PHP, Java**などで作る
- 例：お店のキッチン、店員さん

バックエンドの主な仕事

皆さんの「〇〇したい！」に応える！

1. 「どんなお願いかな？」とリクエストを聞く
2. 必要なら「倉庫」（データベース）へ行く
3. 倉庫から情報を取り出す・しまう
4. お願いされた処理をする
5. 結果をフロントエンドに返す

3. データベース：お店の「倉庫」

- Webサイト・アプリの**「情報」をしまう場所**
- ユーザー情報、商品、記事など、色々なデータを保管
- 整理されていて、**必要な時に素早く取り出せる**
- 例：お店の食材倉庫、商品倉庫

データベースの種類（ざっくり）

- **リレーショナルデータベース (RDB)**:
  - データを**表（テーブル）**みたいに整理するのが得意
  - SQLという言葉で操作する
  - 例：MySQL, PostgreSQL
- **NoSQLデータベース**:
  - データの形がもっと**自由**
  - たくさんのデータを扱うのが得意な場合も
  - 例：MongoDB

バックエンドとデータベースの関係

「店員さん（バックエンド）」と「倉庫（データベース）」

- 店員さん（バックエンド）が、お客さんの注文を受けて**倉庫（データベース）に「〇〇ちょうだい！」とお願い**する
- 倉庫（データベース）から**必要なデータを受け取る**
- そのデータを元に
  料理（処理）をして、お客さんに出す（フロントエンドに渡す）
- 新しい情報を**倉庫にしまう**のも店員さんの仕事

リレーショナルデータベースの特徴

- データを**「テーブル（表）」**の形式で管理
- Excelのシートのようなもの
- データを整理する基本の単位

テーブルの例：お客さんリスト

顧客ID	名前	住所
1	山田 太郎	東京都…
2	佐藤 花子	大阪府…

リレーショナルなワケ：関連付け（リレーション）

- 複数のテーブルを**つなげられる**！
- 共通の項目（キー）を使うよ

例：お客さんリスト と 注文リスト

お客さんリスト

顧客ID	名前	住所
1	山田 太郎	東京都…
2	佐藤 花子	大阪府…

例：お客さんリスト と 注文リスト

注文リスト

注文ID	顧客ID	注文商品	金額
101	**1**	りんご	500
102	**2**	バナナ	300
103	**1**	みかん	400

関連付け（リレーション）のイメージ

`お客さんリスト` と `注文リスト` を **顧客ID** でつなぐ！

山田太郎さん（顧客ID: 1）の注文は？
→ 注文ID: 101 (りんご), 103 (みかん) だ！

なぜリレーショナルデータベースを使うの？

データを正確に保ちやすい

- 変更が一か所で済む
- データに矛盾が起きにくい

必要な情報を簡単に取り出せる

- 関連付けを使って、複数の表からの情報をまとめて取得
- 「このお客さんの全注文履歴」など

扱いやすい

- 多くのRDBは **SQL** という共通言語で操作
- SQLを覚えれば色々使える！

- リレーショナルデータベースは、データを**表（テーブル）**で管理する
- 複数のテーブルを**関連付けられる**のが特徴
- **正確さを保ち、情報の取り出しを簡単に**する
- **SQL**で操作する

RDBの主な構成要素

- **テーブル (Table):**
  - データの集合体（スプレッドシートのシート）
  - 例：「顧客リスト」「商品リスト」
- **カラム / 列 (Column):**
  - データの項目（スプレッドシートの列）
  - 例：「顧客ID」「氏名」「価格」
- **行 / レコード (Row / Record):**
  - 個別のデータ（スプレッドシートの行）
  - 例：「顧客ID: 001, 氏名: 山田太郎...」

テーブルの例：お客さんリスト

顧客ID	名前	住所
1	山田 太郎	東京都…
2	佐藤 花子	大阪府…

RDBの主な構成要素 (キー)

- **主キー (Primary Key):**
  - **各行を一意に識別**するためのカラム
  - 同じ値は存在しない
  - 例：「顧客ID」
- **外部キー (Foreign Key):**
  - **他のテーブルの主キーを参照**するカラム
  - テーブル間の**リレーションシップ**を確立
  - 例：「注文」テーブルの「顧客ID」が「顧客」テーブルの主キーを参照

SQL (Structured Query Language) とは？

- リレーショナルデータベースを**操作するための標準言語**
- データベースへの「命令」を記述
  - データの取得、追加、更新、削除
  - データベース構造の管理

基本的なSQLコマンド

データの操作 (CRUD) + テーブル作成

- `CREATE TABLE` : 新しいテーブルを作成
- `INSERT` : テーブルに新しい行を追加 (Create)
- `SELECT` : テーブルからデータを取得 (Read)
- `UPDATE` : テーブルの既存データを更新 (Update)
- `DELETE` : テーブルのデータを削除 (Delete)

テーブル作成（`CREATE TABLE`）

```sql
CREATE TABLE users (
    user_id INT PRIMARY KEY,          -- ユーザーID（主キー）
    name VARCHAR(100),                -- 名前
    email VARCHAR(255),               -- メールアドレス
    created_at DATETIME               -- 作成日時
);
```

- `INT`, `VARCHAR`, `DATETIME` などは**データ型**

データ追加（`INSERT`）

```
INSERT INTO users (user_id, name, email, created_at)
VALUES (1, '山田太郎', 'yamada@example.com', '2023-01-01 10:00:00');

INSERT INTO users (user_id, name, email, created_at)
VALUES (2, '佐藤花子', 'sato@example.com', '2023-01-01 11:00:00');
```

データ取得（`SELECT`）

- 全てのカラム・全ての行を取得:

```
SELECT * FROM users;
```

- 特定のカラムのみ取得:

```
SELECT name, email FROM users;
```

データ取得（`SELECT` + `WHERE`）

- **条件を指定してデータを取得 (WHERE句):**

```
SELECT * FROM users WHERE user_id = 1; -- user_idが1のユーザー

SELECT * FROM users WHERE name = '佐藤花子'; -- 名前が「佐藤花子」のユーザー
```

データ更新（`UPDATE`）

```sql
UPDATE users
SET email = 'yamada_new@example.com' -- 更新内容
WHERE user_id = 1;                   -- 更新対象の行
```

- `WHERE` 句を忘れると **全ての行が更新** されるので注意！

データ削除（`DELETE`）

```sql
DELETE FROM users WHERE user_id = 2; -- user_idが2のユーザーを削除
```

- `WHERE` 句を忘れると **テーブルの全てのデータが削除** されるので、**特に注意！**

よく使われているデータベース

- **SQLite:**
  - ファイルベースの軽量DB
  - サーバー不要、すぐに始めやすい
- **PostgreSQL や MySQL:**
  - より本格的なDBシステム
  - 学習が進んだら挑戦

- **RDB:** テーブルとリレーションでデータを管理
- **SQL:** RDBを操作する標準言語
- まずは **CREATE TABLE, INSERT, SELECT, UPDATE, DELETE** をマスター！
- **実際に手を動かして練習あるのみ！**

- **FastAPIとは？**
  - PythonでWeb APIを作るための人気フレームワーク。
  - 「速い」「コードが書きやすい」「ドキュメント自動生成」などの特徴があります。
- **SQLiteとは？**
  - 軽量なファイルベースのデータベース。
  - 別にデータベースサーバーを立てる必要がなく、手軽に使えます。

今回作るもの

- **やること:**
  i. データベース (`data.db` ファイル) を準備する。
  ii. データをデータベースに保存できるAPI (`POST /data`) を作る。
  iii. データベースからデータを一覧で取得できるAPI (`GET /data`) を作る。
  iv. 作ったAPIを動かす。

今回使うもの

- **使うもの:**
  - Python
  - FastAPIライブラリ
  - Uvicornライブラリ (FastAPIを動かすサーバー)
  - SQLite

必要なものと実行方法

- **必要なライブラリのインストール:**
  - ターミナルで以下を実行:
    ```
    pip install fastapi uvicorn pydantic
    ```
  - (SQLiteはPythonに標準で含まれています)
- **コードの保存:**
  - Pythonコードを `server.py` として保存。

必要なものと実行方法

- **実行方法:**
  - コードを保存したフォルダでターミナルを開き、以下を実行:

```
python server.py
```

コードを見てみよう - その1

```
from fastapi import FastAPI
# ... 他のimport ...
import sqlite3
import os
import uvicorn

app = FastAPI() # FastAPIアプリを作る！

BASE_DIR = os.path.dirname(__file__)
DB_PATH = os.path.join(BASE_DIR, "data.db") # DBファイルのパス
```

- **import文:** 必要な部品を取り込んでいます。
- `app = FastAPI()`: これが私たちの作るWeb APIの本体です。
- `BASE_DIR`, `DB_PATH`: データベースファイル `data.db` を、このPythonファイルと同じ場所に作成するための設定です。

コードを見てみよう - その2

```python
from pydantic import BaseModel
from typing import List, Optional

class DataBase(BaseModel):
    id: Optional[int] = None
    value_1: str
    value_2: Optional[str] = None
```

- **Pydantic `BaseModel`:**
  - APIが扱うデータの**形（構造）と型（種類）**を定義します。
  - これにより、FastAPIが自動でデータのチェックをしてくれます！
- `id: Optional[int] = None`: ID。整数。なくてもOK（`Optional`）
- `value_1: str`: 必須の文字列データ。
- `value_2: Optional[str] = None`: あってもなくてもOKな文字列データ。

コードを見てみよう - その3

```
import sqlite3 # 再掲

def get_db_connection():
    conn = sqlite3.connect(DB_PATH) # DBファイルに接続
    conn.row_factory = sqlite3.Row # 結果を列名で取り出せるように
    return conn
```

- `get_db_connection()` 関数:
  - データベースファイル (`data.db`) に接続するための共通処理。
  - `conn.row_factory = sqlite3.Row`: 取得したデータを `item['column_name']` のように扱えるようにします。

コードを見てみよう - その3

```
def initialize_db():
    conn = get_db_connection() # DBに接続
    cursor = conn.cursor() # 命令実行用のカーソル
    cursor.execute( # SQLを実行！
        """
        CREATE TABLE IF NOT EXISTS data (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            value_1 TEXT NOT NULL,
            value_2 TEXT
        )
        """
    ) # 'data'というテーブルが無ければ作成
    conn.commit() # 変更を確定
    conn.close() # 接続を閉じる
```

コードを見てみよう - その3

- `initialize_db()` 関数:
  - アプリ起動時、データベースに `data` テーブルが**無ければ**作ります。
  - テーブルの構造:
    - `id`: 自動で増える番号 (`AUTOINCREMENT`), 一意の鍵 (`PRIMARY KEY`)
    - `value_1`: 文字列, 必須 (`NOT NULL`)
    - `value_2`: 文字列

コードを見てみよう - その4

```python
@app.get("/data", response_model=List[DataBase])
def read_data_items():
    conn = get_db_connection() # DB接続
    items = conn.execute("SELECT * FROM data").fetchall() # 全データ取得
    conn.close() # DB接続を閉じる
    # データをDataBaseのリストに変換して返す
    return [DataBase(**dict(item)) for item in items]
```

コードを見てみよう - その4

- `@app.get("/data", ...)`:
  - `/data` というURLに **GET** リクエストが来たら `read_data_items` 関数を実行します。
  - `response_model=List[DataBase]`: 返すのは `DataBase` モデルのリストですよ、と宣言。
- `read_data_items()` 関数:
  - DBから `data` テーブルのすべてのデータを取ってきます (`SELECT * FROM data`)。
  - 取得したデータを `DataBase` モデルのリスト形式に変換して返します。

コードを見てみよう - その5

```python
@app.post("/data", response_model=DataBase, status_code=201)
def create_data_item(item: DataBase):
    conn = get_db_connection() # DB接続
    cursor = conn.cursor() # カーソル取得
    cursor.execute( # SQLを実行！
        "INSERT INTO data (value_1, value_2) VALUES (?, ?)", # データ挿入
        (item.value_1, item.value_2), # ?に値をセット
    )
    conn.commit() # 変更を確定！
    item_id = cursor.lastrowid # 挿入したデータのIDを取得
    conn.close() # DB接続を閉じる
    # 挿入したデータにIDを加えて返す
    return DataBase(id=item_id, value_1=item.value_1, value_2=item.value_2)
```

コードを見てみよう - その5

- `@app.post("/data", ...)`:
  - `/data` というURLに **POST** リクエストが来たら `create_data_item` 関数を実行します。
  - `response_model=DataBase`: 返すのは `DataBase` モデルですよ。
  - `status_code=201`: 作成成功の意味のステータスコードを返す。
- `create_data_item(item: DataBase)` 関数:
  - リクエストボディのデータが自動的に `DataBase` 型の `item` として渡されます。
  - 受け取った `item` の値をDBに挿入します (`INSERT INTO data ...`)。
  - `conn.commit()` で変更を確定！
  - 挿入されたデータの `id` を取得し、それを付けて応答として返します。

コードを見てみよう - その6

```
if __name__ == "__main__":
    initialize_db() # まずDBテーブルが無ければ作る
    uvicorn.run("server:app", host="127.0.0.1", port=8000, reload=True) # サーバー起動！
```

- `if __name__ == "__main__":` ブロック:
  - このスクリプトが**直接実行されたとき**だけ動く部分。
- `initialize_db()`: アプリ起動前にDBテーブルを作成。
- `uvicorn.run(...)`:
  - Uvicornサーバーを起動し、FastAPIアプリ (`server.py` の `app`) を動かします。
  - `host="127.0.0.1", port=8000`: 自分のPCの8000番ポートで待ち受け。
  - `reload=True`: コード変更を自動で反映 (開発用)。

まとめ

- データベースは、データを整理して保存するための「倉庫」
- リレーショナルデータベースは、データを表形式で管理し、関連付けができる
- SQLを使ってデータの操作ができる
- FastAPIを使って、データベースと連携するWeb APIを作成しました